Clase 3 — 31.10.2025

#JAVA

👨‍🏫 *Profesor: Joan Salvador Gordi Ortega*
📘 Programación Multimedia y Dispositivos Móviles
🗓️ Clase 3 — 31/10/2025
🎯 Tema: **Repaso de la Programación Orientada a Objetos (POO) en Java II**

5️⃣ Conceptos POO para Android

El profesor utiliza un ejemplo “de empresa” en Java puro (sin Android) para repasar:

- Clases abstractas
- Herencia
- Polimorfismo
- Interfaces

y luego conectar estos conceptos con lo que veremos en Android (Activities, Listeners, etc.).

Todo se hace en **Eclipse**, porque:

- es gratuito,
- es muy usado en entorno empresarial,
- y es suficiente para proyectos Java sencillos.

Perfecto. Amplío **mucho más** la teoría, argumento cada punto, y mantengo exactamente la estructura que has indicado.
Aquí tienes la versión extendida y profundamente explicada:

5.1 Creación del proyecto en Eclipse

En esta parte el profesor no solo muestra los pasos técnicos para crear un proyecto, sino que explica **por qué se organiza así el entorno** y cómo esta estructura se relaciona con buenas prácticas en desarrollo profesional.

1. `File → New → Java Project`

Este es el punto de partida para levantar un proyecto Java puro dentro de Eclipse.
El profesor comenta que Eclipse es un entorno especialmente adecuado para:

- proyectos educativos,
- prácticas profesionales sencillas,
- y desarrollo Java empresarial clásico (backend con Java SE o Java EE).

Es un IDE muy utilizado en empresas por su equilibrio entre:

- **costo** (gratuito),
- **funcionalidad**,
- **estabilidad**,
- y la gran cantidad de plugins existentes.

En comparación, IntelliJ o Android Studio están más orientados a entornos concretos (Kotlin/Android, proyectos modernos…), pero Eclipse sigue siendo el estándar en proyectos Java de muchas organizaciones.

---

2. Indica un nombre, por ejemplo: `PMPDM.P01.Empleados`

El profesor subraya la importancia de usar nombres claros y que den contexto:

- `PMPDM` → Programación Multimedia y Dispositivos Móviles
- `P01` → Práctica o Proyecto 01
- `Empleados` → Tema principal

Esto imita la estructura de proyectos reales, donde los nombres deben indicar:

- el módulo,
- la funcionalidad,
- o el objetivo concreto.

Una nomenclatura pobre lleva al caos en proyectos grandes.

---

3. Desmarcar: `Create module-info.java file`

Esta opción aparece desde Java 9, que introdujo el sistema de módulos (JPMS — Java Platform Module System).

El profesor **la desmarca a propósito** porque:

- los módulos complican innecesariamente proyectos educativos,
- generan más ficheros y reglas de las necesarias,
- y muchas librerías externas siguen sin soportarlos del todo.

En cursos introductorios, usar módulos provoca:

- errores de accesibilidad,
- problemas al importar paquetes,
- conflictos entre módulos,
- y una curva de aprendizaje no necesaria.

Por eso, se trabaja en modo **Java clásico**, lo que simplifica el trabajo y facilita concentrarse en POO.

---

4. Dentro de `src`, crear un package:

```
gestion.empleados
```

El nombre sigue la convención Java de **dominio invertido** (aunque aquí se simplifica):

- En proyectos reales sería algo como:
  `com.miempresa.gestion.empleados`

El package agrupa todas las clases relacionadas dentro del mismo módulo funcional:

- Permite organizar mejor el código.
- Evita colisiones de nombres entre clases.

- Facilita la reutilización del software.

El profesor insiste en que, igual que Android organiza su código en packages (`com.android.settings` , `androidx.lifecycle` , etc.), los proyectos Java deben estructurarse igual.

---

5. Dentro del package se crean las clases:

- `Empleado`
- `EmpleadoGerente`
- `EmpleadoPermanente`
- `EmpleadoTemporal`
- `IBonificacion`
- `Accionista`
- `Main`

Esta estructura busca replicar un escenario real:

- una clase base (`Empleado`),
- especializaciones por tipo de empleado (herencia),
- una interfaz que define comportamiento común opcional (`IBonificacion`),
- y una clase externa pero relacionada (`Accionista`),
- junto a una clase de pruebas donde se instancian y manipulan los objetos (`Main`).

El profesor enfatiza que esta forma de organizar el código es casi idéntica a la que se utiliza en Android:

- Activities, Fragments, Interfaces y Helpers viven en packages propios.
- La estructura modular permite localizar más rápido clases y responsabilidades.

Herramientas clave de Eclipse

Estas herramientas automatizan tareas repetitivas y favorecen un código limpio.

`Source → Generate Getters and Setters`

Permite generar automáticamente:

```
public int getId() { ... }
public void setId(int id) { ... }
```

Ventajas:

- rapidez,
- evita errores de escritura,
- asegura consistencia,
- y respeta el principio de encapsulación.

`Source → Generate toString()`

Genera un método que devuelve una representación legible del objeto:

```
@Override
public String toString() {
```

```
    return "Empleado{id=1, nombre='Juan', salarioBase=1200}";
}
```

Esto es fundamental para depurar y para imprimir objetos en consola o logs.

`Source → Override/Implement Methods`

Permite insertar automáticamente:

- métodos abstractos obligatorios,
- métodos de interfaces,
- sobrescrituras (`@Override`),
- y funciones heredadas.

Ejemplo generado automáticamente:

```
@Override
public double calcularSalario() {
    // TODO Auto-generated method stub
    return 0;
}
```

El profesor insiste en que esto ayuda a:

- asegurar que la firma del método coincide exactamente con la original,
- evitar errores de compilación,
- y trabajar más rápido en proyectos profesionales.

5.2 Clase base `Empleado` — Encapsulación y abstracción

La clase `Empleado` es el núcleo conceptual del ejemplo del profesor. Su diseño sirve para ilustrar varios principios fundamentales de la Programación Orientada a Objetos **tal como se aplican en Java moderno y, más adelante, en Android**.

El profesor insiste en que esta clase **no es simplemente una clase**, sino un **patrón de diseño en sí misma**:
una *superclase abstracta que define un contrato común y aplica encapsulación sobre atributos críticos*.

5.2.1 Encapsulación: el principio que protege la estabilidad del sistema

Atributos privados

```
private int id;
private String nombre;
private double salarioBase;
```

El profesor explica que esta decisión no es estética: es **arquitectura**.

Ventajas profundas de encapsular atributos:

1. **Evita “corrupción del estado”**
   Ninguna clase externa puede modificar directamente los valores internos.
   Esto evita bugs donde dos partes del código manipulan el mismo atributo de forma inesperada.

2. **Permite controlar la lógica de asignación**
   Un setter puede validar, corregir o rechazar valores incorrectos.

   Ejemplo realista que menciona el profesor:

```
public void setSalarioBase(double salarioBase) {
    if(salarioBase < 0) throw new IllegalArgumentException("El salario no puede ser
negativo");
    this.salarioBase = salarioBase;
}
```

3. **Permite cambiar la implementación interna sin afectar al exterior**
   Incluso podríamos cambiar el tipo de una variable (por ejemplo, `double` → `BigDecimal` para cálculos financieros)
   sin modificar el código de las clases que la usan.
4. **Es obligatorio para cumplir los principios SOLID**
   Especialmente *Single Responsibility* y *Open/Closed*, muy relevantes en código de empresa.

5.2.2 Getters y Setters: más que accesores

Aunque muchos alumnos los ven como algo “mecánico”, el profesor recalca que los getters/setters son:

- puertas de acceso controlado a los datos,
- puntos ideales para añadir validaciones,
- mecanismos para integrarse con frameworks que usan JavaBeans,
- y herramientas indispensables en Android, donde muchos componentes usan setters para actualizar estado (por ejemplo, cambiar texto de un `TextView`).

También los relaciona con prácticas reales:

- Registro de auditoría:
  se puede loguear cada cambio importante.
- Lazy-loading:
  en getters se puede calcular un valor bajo demanda.
- Seguridad:
  se puede impedir que ciertas propiedades se modifiquen si el objeto está en cierto estado.

5.2.3 Método `calcularSalario()` y el principio de abstracción

Este es el elemento estrella de la clase:

```
public abstract double calcularSalario();
```

¿Por qué es abstracto?

El profesor explica que la abstracción modela **conceptos incompletos** que deben ser concretados por los tipos reales.

En la vida real:

- no existe “un empleado genérico” cuyo salario se calcula siempre igual,
- el salario depende de la categoría del trabajador,
- por lo tanto **la superclase no puede dar una implementación válida**.

¿Qué logra este diseño?

1. **Forzar a las subclases a definir su comportamiento**
   No se puede olvidar implementar el cálculo del salario:
   el compilador obligará a definirlo.
2. **Evitar comportamientos incorrectos**
   Si pusiéramos un valor por defecto en la superclase, sería muy probable que alguien lo reutilizara sin querer.
3. **Promover un diseño centrado en comportamiento**
   El profesor recalca que la POO moderna no consiste solo en "tener clases con atributos",
   sino en **definir responsabilidades claras**.
4. **Permitir polimorfismo real**
   El método abstracto es la clave para que el siguiente código funcione:

   ```
   Empleado e = new EmpleadoGerente();
   double sueldo = e.calcularSalario();
   ```

   Aunque la variable es de tipo `Empleado`, la ejecución depende del objeto concreto.

5.2.4 La clase `Empleado` como arquetipo profesional

El profesor aprovecha para destacar que clases como `Empleado` representan un patrón muy común en la industria:

- una superclase abstracta,
- con atributos comunes,
- con métodos abstractos,
- y con comportamiento parcialmente implementado.

Este patrón aparece en infinidad de APIs:

API / Entorno	Clase base abstracta	Subclases
Android	`Activity`	`MainActivity`, `LoginActivity`, etc.
Android	`View`	`Button`, `TextView`, `ImageView`
Java	`InputStream`	`FileInputStream`, `BufferedInputStream`
JavaFX	`Application`	aplicaciones gráficas
JDBC	`Connection`	implementaciones de Oracle, MySQL, etc.

El profesor establece la analogía:

> "Si entiendes por qué `Empleado` es abstracta, entenderás por qué `Activity` también lo es, y por qué no puedes crear una Activity sin sobrescribir métodos como `onCreate()`."

5.2.5 El método `toString()` y su importancia en depuración

```
@Override
public String toString() {
    return "Empleado{" +
            "id=" + id +
            ", nombre='" + nombre + '\'' +
```

```
            ", salarioBase=" + salarioBase +
            '}';
}
```

Aunque pueda parecer accesorio, el profesor lo considera fundamental:

- Permite imprimir el estado del objeto fácilmente.
- Es extensamente usado en depuración y logging.
- Evita tener que inspeccionar objetos complejos manualmente.
- Refuerza el hábito profesional de hacer clases legibles, no cajas negras.

En entornos empresariales, `toString()` suele utilizarse incluso para:

- auditorías,
- serialización temporal,
- trazas en sistemas distribuidos,
- análisis de errores.

5.2.6 La clase `Empleado` como contrato conceptual

Finalmente, el profesor insiste en que esta clase representa más que código:

- es una **plantilla lógica**,
- un **modelo de negocio**,
- una **abstracción de un concepto real**.

Define:

- qué datos “mínimos” tiene un empleado,
- qué acción importante debe poder realizar (calcular salario),
- y qué partes son comunes y cuáles cambian entre tipos concretos.

Este enfoque es exactamente el que se utiliza en desarrollo Android cuando se diseña:

- una jerarquía de vistas,
- un patrón MVP/MVVM,
- una arquitectura basada en repositorios,
- o un conjunto de Activities/Fragments.

5.3 Analogía con `Figura2D` — Comprender la abstracción a través de un ejemplo universal

El profesor introduce un segundo ejemplo (la jerarquía `Figura2D`) porque es mucho más visual y conceptual que el ejemplo de empleados. Sirve para que incluso quien no domina el contexto empresarial pueda **entender de forma intuitiva** qué significa una clase abstracta y por qué existe.

Este ejemplo suele aparecer en libros, entrevistas y cursos universitarios porque encapsula perfectamente estos conceptos:

- abstracción,
- herencia,
- polimorfismo,

- reutilización de código,
- especialización progresiva.

Vamos a ampliarlo con detalle.

5.3.1 Clase abstracta `Figura2D`: el modelo general

```
public abstract class Figura2D {

    protected double base;
    protected double altura;

    public Figura2D(double base, double altura) {
        this.base = base;
        this.altura = altura;
    }

    public abstract double calcularArea();

    public void mostrarInformacion() {
        System.out.println("Figura de base = " + base + " y altura = " + altura);
    }
}
```

Elementos clave del diseño

1. Atributos comunes (`base`, `altura`)

Una figura geométrica siempre tiene dimensiones propias. Son propiedades que existen en prácticamente todas las figuras 2D, así que tiene sentido colocarlas en la superclase.

Aquí el profesor introduce una idea importante:

> "La abstracción no significa ausencia de atributos.
> Significa ausencia de *comportamiento concreto*".

Una clase abstracta puede tener:

- atributos normales,
- métodos totalmente implementados,
- constructores,
- y por supuesto, métodos abstractos.

---

2. Constructor con parámetros

Esto demuestra que una clase abstracta:

- sí puede tener constructor,
- pero se invoca **desde las subclases**, no desde un `new`.

Ejemplo:

```
public Rectangulo(double base, double altura) {
    super(base, altura);
```

```
}
```

---

3. Método abstracto: `calcularArea()`

Aquí está la esencia:

```
public abstract double calcularArea();
```

El área de una figura depende del tipo de figura.
No existe un área genérica.

Ejemplos reales:

- Triángulo → (base × altura) / 2
- Rectángulo → base × altura
- Rombo → (D × d) / 2
- Círculo → π × r² (ni siquiera usa base/altura)

Por eso esta lógica **no puede estar en la superclase**, ya que sería incorrecta para la mayoría de figuras.

La abstracción impone una regla:

> "Si heredas de `Figura2D`, debes definir cómo calcular el área."

---

4. Método concreto: `mostrarInformacion()`

Esto ilustra que la abstracción **no prohibe implementar métodos comunes**:

```
public void mostrarInformacion() {
    System.out.println("Figura de base = " + base + " y altura = " + altura);
}
```

Este método:

- existe para todas las figuras,
- se ejecuta igual independientemente del tipo,
- forma parte del comportamiento común reutilizable.

Esta es la esencia de la herencia:

- **Código general** → en la superclase
- **Código específico** → en las subclases

5.3.2 Subclase `Triangulo`

```
public class Triangulo extends Figura2D {

    public Triangulo(double base, double altura) {
        super(base, altura);
    }

    @Override
    public double calcularArea() {
```

```
        return (base * altura) / 2;
    }
}
```

El profesor resalta varios conceptos importantes:

- Hereda atributos → no se vuelven a declarar.
- Hereda métodos → `mostrarInformacion()` ya existe.
- Está obligado a implementar `calcularArea()`.

Esto es especialización:
el triángulo concreta lo que `Figura2D` dejó pendiente.

5.3.3 Subclase `Rectangulo`

```
public class Rectangulo extends Figura2D {

    public Rectangulo(double base, double altura) {
        super(base, altura);
    }

    @Override
    public double calcularArea() {
        return base * altura;
    }
}
```

De nuevo, el profesor muestra que:

- dos clases pueden heredar del mismo padre,
- pero aportar comportamientos completamente diferentes,
- y aun así compartir la misma interfaz pública.

Esto es fundamental en el diseño orientado a objetos:

- estructuras diferentes,
- comportamientos personalizados,
- pero **misma familia conceptual**.

5.3.4 Polimorfismo aplicado a figuras

El profesor demuestra cómo usar estas clases:

```
Figura2D f1 = new Triangulo(10, 5);
Figura2D f2 = new Rectangulo(10, 5);

System.out.println(f1.calcularArea());
System.out.println(f2.calcularArea());
```

El polimorfismo activo permite:

- guardar objetos distintos bajo una misma referencia,
- invocar métodos compartidos (`calcularArea()`, `mostrarInformacion()`),

- sin preocuparte del tipo concreto.

Esto modela una idea muy poderosa:

> "No importa *qué tipo* de figura tengo,
> importa *what it can do*: calcular el área."

Este mismo patrón se repite en Android constantemente:

- `View v = new Button(context);`
- `Drawable d = new GradientDrawable();`
- `LayoutManager lm = new LinearLayoutManager(...)`

5.3.5 Error común del alumno: "Si la clase tiene propiedades… ¿ya no puede ser abstracta?"

El profesor recibe esta duda en clase y la desmonta con claridad:

> **Sí puedes tener propiedades en una clase abstracta.**
> **Lo que NO puedes tener es un método abstracto sin implementar en una clase normal.**

Explicación ampliada:

- La abstracción no afecta a atributos.
- La abstracción afecta a **comportamientos incompletos**.
- Si una clase define un método abstracto, la clase debe ser abstracta.
- Los atributos pueden existir sin problema, incluso aunque la clase no se pueda instanciar.

Ejemplo:

```
public abstract class Figura3D {
    protected String color;   // atributo normal
    public abstract double volumen();   // método abstracto
}
```

Todo esto es válido.

5.3.6 No se puede instanciar una clase abstracta

```
Figura2D f = new Figura2D(10, 20); // ❌ Error de compilación
```

El profesor insiste en que esto no es una limitación arbitraria, sino un mecanismo de protección:

- Si se permitiera instanciarla, no habría implementación de `calcularArea()`.
- Java no sabría qué hacer cuando se invocara ese método.
- Se rompería el contrato de comportamiento.

En términos de diseño:

> "Una clase abstracta describe *qué es*,
> pero no describe completamente *cómo funciona*."

5.3.7 Resumen conceptual del profesor

El ejemplo de `Figura2D` deja claras estas ideas:

- La **abstracción** representa modelos incompletos pero conceptualmente válidos.
- La **herencia** especializa comportamientos.
- El **polimorfismo** permite tratar objetos diferentes de forma uniforme.
- La **reutilización de código** evita duplicación innecesaria.
- El **diseño limpio** requiere separar:
  - comportamiento común → superclase
  - comportamiento específico → subclases

Y sobre todo:

> "Una clase abstracta no es para instanciar,
> es para **ser extendida**."

5.4 Herencia en la jerarquía de Empleados — Especialización y polimorfismo básico

Tras construir la clase abstracta `Empleado`, el profesor pasa a explicar cómo se crean **subclases concretas** que extienden de ella.
Este apartado sirve para introducir tres conceptos clave:

1. **Herencia (extends)**
2. **Especialización progresiva**
3. **Polimorfismo a través de la superclase común**

El objetivo es demostrar que la POO no consiste en tener "muchas clases", sino en tener **clases relacionadas jerárquicamente**, donde cada una aporta algo nuevo o redefine un comportamiento.

Vamos por partes.

5.4.1 Por qué heredar de `Empleado`

El profesor comienza justificando la herencia desde el punto de vista del diseño:

> "Si varias clases comparten atributos y comportamientos, lo correcto es que hereden de una clase padre que represente su esencia común."

En el ejemplo, todos los empleados de la empresa:

- tienen un `id`,
- tienen un `nombre`,
- tienen un `salarioBase`,
- y **todos deben saber calcular su salario**.

Esta última parte es clave:

- la fórmula cambia según el tipo de empleado,
- pero **la existencia del método es común**.

En otras palabras:

> “No comparten la *implementación*, pero sí la *responsabilidad*.”

Ese es el motivo por el que `Empleado` define un método abstracto:

```
public abstract double calcularSalario();
```

Y ese método actúa como un *contrato* que todas las subclases deben cumplir.

5.4.2 Subclase `EmpleadoGerente` — Caso de implementación simple

La primera subclase que crea el profesor es:

```
public class EmpleadoGerente extends Empleado {

    @Override
    public double calcularSalario() {
        return getSalarioBase() + 300; // plus de gerente
    }
}
```

Puntos clave explicados por el profesor

1. **La clase no es abstracta** porque implementa el método abstracto de la superclase.
2. **Heredamos los atributos**, así que no deben redeclararse.
3. El uso de `getSalarioBase()` demuestra el principio de encapsulación.
4. Añadir un plus es un ejemplo de **especialización sencilla**.

En la práctica profesional, esta especialización podría depender de:

- variables internas,
- rendimiento,
- incentivos,
- antigüedad,
- factores externos.

Pero para la clase, basta con entender la mecánica.

5.4.3 Subclase `EmpleadoTemporal` — Caso sin bonificaciones

```
public class EmpleadoTemporal extends Empleado {

    @Override
    public double calcularSalario() {
        return getSalarioBase(); // cobra solo el salario base
    }
}
```

Este ejemplo sirve para demostrar que:

- no todas las clases necesitan lógica compleja,
- pero **todas están obligadas a definir el método**,

- y no es necesario que todas compartan propiedades o bonificaciones.

Este punto es importante porque anticipa el problema de “ensuciar la herencia”, que se resolverá con interfaces más adelante.

5.4.4 Subclase `EmpleadoPermanente` — Caso con bonificación (antes de añadir interfaz)

El profesor presenta esta clase incluso antes de introducir `IBonificacion` porque quiere demostrar cómo se acumulan responsabilidades:

```
public class EmpleadoPermanente extends Empleado {

    @Override
    public double calcularSalario() {
        // Implementación provisional
        return getSalarioBase() * 1.20;
    }
}
```

Aquí se ve:

- que no es necesario repetir atributos,
- que la subclase aporta lógica propia,
- que `Empleado` actúa como molde.

Antes de introducir interfaces, esta clase ya anticipa que:

- algunas subclases tendrán bonificación,
- otras no,
- y que **no todas comparten la misma estructura lógica**.

Esto prepara el terreno para explicar por qué una clase base NO debe cargar con métodos que solo algunos hijos necesitan.

5.4.5 Uso de `@Override` como buena práctica obligatoria

El profesor insiste en que:

- siempre debe usarse `@Override`,
- incluso cuando Java no lo exige explícitamente.

¿Motivos?

1. **El compilador verifica que el método existe en la superclase.**
   Sin `@Override`, podrías escribir mal el nombre y crear un método nuevo por error.
2. **Es más legible**, especialmente en jerarquías grandes.
3. En Android, muchas veces se depende de métodos del ciclo de vida (`onCreate`, `onClick`, etc.), y escribir uno incorrectamente provocaría fallos difíciles de detectar.

Ejemplo de error típico evitado con `@Override`:

```
public double calcularsalario() { ... } // ❌ mal escrito
```

Con Override:

```
@Override
public double calcularsalario() {}
// ❌ error inmediato: no coincide con método abstracto
```

5.4.6 Polimorfismo básico con `Empleado`

Una vez definidas las subclases, el profesor demuestra la potencia del polimorfismo:

```
Empleado e1 = new EmpleadoGerente();
Empleado e2 = new EmpleadoPermanente();
Empleado e3 = new EmpleadoTemporal();
```

Aunque todas las variables son de tipo `Empleado`,
el comportamiento en tiempo de ejecución **depende de la clase real** del objeto.

Ejemplo:

```
System.out.println(e1.calcularSalario()); // usa la lógica de Gerente
System.out.println(e2.calcularSalario()); // usa la lógica de Permanente
System.out.println(e3.calcularSalario()); // usa Temporal
```

Conceptos clave que subraya el profesor:

1. Llamamos al mismo método, pero ejecuta lógica diferente.

Esto es polimorfismo dinámico.

2. Podemos almacenar todos los empleados en una misma estructura:

```
Empleado[] plantilla = {
    new EmpleadoGerente(),
    new EmpleadoPermanente(),
    new EmpleadoTemporal()
};
```

3. La interfaz común es la superclase.

Esto se parece mucho a cómo trabaja Android:

```
View v = new Button(context);
View v2 = new TextView(context);
View v3 = new ImageView(context);
```

5.4.7 Beneficios profesionales del uso correcto de herencia

El profesor insiste en que esta estructura no es solo pedagógica.
Representa un patrón de diseño utilizado constantemente en el mundo real:

✔ Reutilización de código

La superclase contiene:

- atributos comunes,
- lógica compartida,
- validaciones,
- métodos representativos como `toString()`.

Las subclases solo contienen lo necesario.

✔ Cohesión y mantenimiento

Si mañana añadimos un atributo nuevo a todos los empleados:

```
private String departamento;
```

solo se añade en un sitio: `Empleado`.

Todas las subclases lo heredan automáticamente.

✔ Diseño más semántico

La jerarquía se convierte en un mapa conceptual del negocio de la empresa:

```
Empleado
 ├── EmpleadoGerente
 ├── EmpleadoPermanente
 └── EmpleadoTemporal
```

Esto es mucho más expresivo que tener clases aisladas y repetidas.

5.4.8 Resumen conceptual del profesor

El objetivo de esta parte es fijar la idea de herencia:

- La clase base define una **identidad común**.
- Las subclases añaden **comportamiento concreto**.
- El polimorfismo permite tratar objetos distintos como iguales.
- La abstracción asegura que las subclases implementen lo necesario.
- La jerarquía mantiene el código organizado y escalable.

Cerrando esta parte, el profesor dice:

> “Herencia no significa copiar cosas del padre.
> Significa **ser una especialización del padre**.”

5.5 El problema de “ensuciar la herencia” — Diseño incorrecto en jerarquías de clases

En este punto, el profesor introduce un concepto de diseño fundamental que suele causar errores incluso en programadores con experiencia:
la **herencia sucia** (*dirty inheritance*).

Este concepto aparece cuando se añaden a una superclase métodos o atributos que **no deberían ser comunes** a todas sus subclases.

Es decir:

> “Estás introduciendo en la clase padre comportamientos que no son universales para todos los hijos.”

Esto rompe principios esenciales de la POO y conduce a código frágil, incoherente y difícil de mantener.

5.5.1 Cómo aparece este problema en la práctica

Tras haber creado las clases:

- `EmpleadoGerente`
- `EmpleadoPermanente`
- `EmpleadoTemporal`

el profesor plantea un escenario real:

- algunos empleados tienen bonificación,
- otros no,
- y además puede haber actores externos (como `Accionista`) que también tengan bonificación.

En primera instancia, alguien podría pensar:

> “Pues añado el método `calcularBonificacion()` directamente en `Empleado` y ya lo heredarán todos”.

Ejemplo:

```
public class Empleado {
    ...

    public double calcularBonificacion() {
        return 0; // valor por defecto
    }
}
```

Este es el típico error que el profesor quiere evitar.

5.5.2 ¿Qué está mal en añadir métodos no universales a la clase base?

El profesor identifica varios problemas graves:

---

1. Rompe la semántica de la jerarquía

Si un método no es común a todas las subclases, **no pertenece a la superclase**.

Ejemplo:

- `EmpleadoTemporal` no debe tener bonificación,
- `EmpleadoPermanente` sí,
- `Gerente` también,
- `Accionista` también, sin ser ni siquiera empleado.

Por tanto, incluirlo como método base:

```
public double calcularBonificacion()
```

está conceptualmente mal.

---

2. Obligamos a subclases a tener métodos que no necesitan

Ejemplo:

```
EmpleadoTemporal et = new EmpleadoTemporal();
et.calcularBonificacion(); // ??? ¿qué significado real tiene?
```

Esto provoca:

- lógica artificial,
- valores por defecto sin sentido,
- confusión sobre el propósito del método.

---

3. Crea comportamientos falsos o incorrectos

Si un método devuelto por defecto es:

```
return 0;
```

las clases que NO deberían tenerlo **parecen tenerlo**, aunque su valor sea “cero”.
Esto distorsiona el modelo real del negocio.

Ejemplo: `Accionista` no es empleado, pero sí tiene bonificación.
No hay forma coherente de meterlo en la jerarquía si el método está en `Empleado`.

---

4. Hace imposible reutilizarlo fuera de la jerarquía

Si la bonificación está dentro de `Empleado`, solo las clases que extiendan de `Empleado` podrán usarlo.
Pero hay casos como `Accionista` donde el comportamiento existe **sin** compartir su naturaleza.

> “La herencia modela lo que un objeto *es*;
> las interfaces modelan lo que un objeto *puede hacer*.”

---

5. Violación del Principio de Sustitución de Liskov (LSP)

El LSP indica que:

> “Una subclase debe poder sustituir a su superclase sin romper la lógica.”

Si `Empleado` define `calcularBonificacion()`,
entonces *todos* los empleados deben tener una bonificación válida.

Pero esto **no se cumple** en empleados temporales.

---

6. Violación del Principio de Responsabilidad Única (SRP)

La superclase `Empleado` pasaría a manejar:

- salario base,
- cálculo de salario,

- nombre e id,
- **bonificaciones** (responsabilidad extra y no universal).

Una clase base debe tener una responsabilidad clara y coherente.
Introducir en ella métodos no compartidos la convierte en un “cajón de sastre”.

5.5.3 Ejemplo visual del “ensuciado”

Un modelo mal diseñado terminaría siendo algo así:

```
Empleado
 ├── EmpleadoGerente
 ├── EmpleadoPermanente
 ├── EmpleadoTemporal   (no tiene bonificación, pero la hereda igual)
 └── Accionista???       (no puede estar aquí pero necesita bonificación)
```

Esto demuestra que:

- la bonificación NO pertenece a la jerarquía,
- pero sí pertenece a algunas de las clases.

Por tanto, la bonificación es un **comportamiento transversal**,
no una característica jerárquica.

5.5.4 La solución correcta: separar comportamiento de identidad

El profesor explica el principio:

> “La herencia se usa para modelar ‘qué es algo’.
> Las interfaces se usan para modelar ‘qué sabe hacer algo’.”

La bonificación NO es una característica de:

- todos los empleados,
- ni de un tipo concreto de empleado,
- ni de una jerarquía típica.

Es una acción que ciertos objetos **saben realizar**.

Por eso debe estar representada mediante una **interfaz**, no mediante herencia.

5.5.5 Alternativa correcta: interfaz `IBonificacion`

```java
public interface IBonificacion {
    double calcularBonificacion();
}
```

De esta forma:

- cada clase que necesita bonificación la implementa,
- cada clase que no la necesita simplemente NO la implementa,
- no se ensucia la superclase,

- no se fuerza a nadie a tener un comportamiento que no corresponde.

Ejemplos correctos:

```
public class EmpleadoPermanente extends Empleado implements IBonificacion {
    @Override
    public double calcularBonificacion() { return 1.20; }
}
```

```
public class Accionista implements IBonificacion {
    @Override
    public double calcularBonificacion() { return 500; }
}
```

```
public class EmpleadoTemporal extends Empleado {
    // No implementa IBonificacion → correcto
}
```

Gracias a esta separación de responsabilidades, el diseño es:

- coherente,
- mantenible,
- extensible,
- realista,
- y correcto desde el punto de vista de la arquitectura.

5.5.6 Enseñanza final del profesor: “Herencia limpia”

Una jerarquía bien diseñada:

- pone en la superclase SOLO lo que es común a todos,
- pone en las subclases lo que es exclusivo de cada tipo,
- usa interfaces para comportamientos opcionales,
- evita métodos por defecto sin sentido,
- permite añadir nuevas clases sin romper el diseño completo.

El profesor resume esta idea con una frase clave:

> “No metas en la clase padre lo que no sea universal.
> Si no lo comparten todos, NO va ahí.”

5.6 La interfaz `IBonificacion` — Comportamientos opcionales y polimorfismo transversal

Tras explicar el problema de “ensuciar la herencia”, el profesor introduce la solución correcta:
usar una **interfaz** para representar un comportamiento que *NO es universal*, pero que sí es compartido por ciertos objetos.

Este punto es crucial porque introduce el segundo gran mecanismo del polimorfismo en Java:

- **Polimorfismo por herencia** → cuando varias subclases comparten un padre.

- **Polimorfismo por interfaz** → cuando varias clases comparten un comportamiento, aunque no compartan jerarquía.

Aquí empieza la parte **más importante** del diseño profesional en POO moderna.

5.6.1 ¿Qué es una interfaz y por qué se usa aquí?

El profesor define una interfaz como:

> "Un contrato. Un conjunto de métodos que una clase promete implementar."

No contiene propiedades, ni lógica interna, ni constructores.
Solo define *qué debe hacer* la clase que la implemente.

Ejemplo básico:

```
public interface IBonificacion {
    double calcularBonificacion();
}
```

Esto obliga a que **cualquier clase** que implemente la interfaz proporcione una implementación concreta del método.

La interfaz NO dice:

- cuánto es la bonificación,
- cómo se calcula,
- si depende del salario base o no,
- si es un porcentaje o una cantidad fija.

Solo impone:

> "Si implementas `IBonificacion`, debes saber calcular una bonificación."

5.6.2 Diferencia esencial: identidad vs comportamiento

El profesor remarca una distinción muy importante:

Herencia (extends) modela identidad

- Un `EmpleadoPermanente` **es** un `Empleado`.
- Un `Triangulo` **es** una `Figura2D`.
- Un `Button` **es** una `View`.

Estas relaciones son **ontológicas**, reflejan *qué es* cada clase.

Interfaces (implements) modelan comportamiento

- Un `EmpleadoPermanente` **puede tener** bonificación.
- Un `Accionista` **puede tener** bonificación.
- Un `EmpleadoTemporal` **no la tiene**, por lo tanto **no implementa la interfaz**.

La interfaz no obliga a la subclase a pertenecer a la jerarquía de empleados.
Por eso es perfecta para representar:

- comportamientos extras,
- roles específicos,
- capacidades opcionales.

En Android es igual:

- `OnClickListener` → “algo que sabe responder a un click”.
- `Runnable` → “algo que sabe ejecutarse en un hilo”.
- `TextWatcher` → “algo que sabe reaccionar cuando cambia un texto”.

Ninguno de ellos pregunta por la jerarquía de la clase. Solo exige el comportamiento.

5.6.3 Implementación de la interfaz en clases concretas

El profesor muestra cómo un `EmpleadoPermanente` la implementa:

```
public class EmpleadoPermanente extends Empleado implements IBonificacion {

    @Override
    public double calcularBonificacion() {
        return 1.25; // 25% extra sobre el salario
    }

    @Override
    public double calcularSalario() {
        return getSalarioBase() * calcularBonificacion();
    }
}
```

Observaciones clave:

- Usa `implements` (no `extends`).
- La clase sigue siendo una subclase de `Empleado`.
- Añade el comportamiento opcional de la bonificación.
- La implementación es flexible y específica.

5.6.4 Caso especial: `Accionista` — el ejemplo clave

Aquí el profesor golpea fuerte con un ejemplo decisivo:

```
public class Accionista implements IBonificacion {

    @Override
    public double calcularBonificacion() {
        return 500; // bonificación independiente del salario
    }
}
```

Este ejemplo demuestra:

- `Accionista` **no es** un `Empleado`.
- No tiene `salarioBase`, ni `id`, ni contrato laboral.

- Pero **sí participa en el sistema de bonificaciones**.

Si hubiéramos puesto la bonificación en la clase base `Empleado`, este caso sería imposible.

Por eso la interfaz es la herramienta correcta:

- flexible,
- neutral,
- desacoplada de la jerarquía,
- universal para cualquiera que la implemente.

5.6.5 El profesor introduce el concepto de “polimorfismo transversal”

A diferencia del polimorfismo por herencia (“todas son Empleado”),
aquí tenemos un polimorfismo que atraviesa jerarquías completamente diferentes.

Ejemplo:

```
IBonificacion b1 = new EmpleadoPermanente();
IBonificacion b2 = new EmpleadoGerente();
IBonificacion b3 = new Accionista();
```

No importa cuál es la clase real.
Son polimórficos porque todos pueden hacer lo mismo:

```
b1.calcularBonificacion();
b2.calcularBonificacion();
b3.calcularBonificacion();
```

Y esto es posible SIN herencia compartida.

El profesor explica que este tipo de polimorfismo es uno de los pilares en frameworks modernos:

- Spring usa interfaces para inyección de dependencias.
- Android usa interfaces para eventos y callbacks.
- JavaFX usa interfaces para listeners.
- JDBC implementa interfaces para controladores de bases de datos.

5.6.6 Combinación de herencia + interfaz

La arquitectura que sale del ejemplo del profesor es así:

```
Empleado (abstract)
 ├── EmpleadoGerente (puede tener bonificación)
 ├── EmpleadoPermanente (tiene bonificación)
 └── EmpleadoTemporal (no tiene bonificación)

Accionista (no es Empleado, pero tiene bonificación)
```

INTERFAZ TRANSVERSAL:

```
IBonificacion
 ├── EmpleadoPermanente
```

```
  ├── EmpleadoGerente
  └── Accionista
```

Esto genera dos ejes:

- **Eje jerárquico** → relación “es un”
- **Eje de comportamiento** → relación “puede hacer”

Este esquema es extremadamente realista en sistemas complejos y en Android.

5.6.7 Método `TratamientoBonificaciones()` — Ejemplo completo del profesor

```java
private static void TratamientoBonificaciones(IBonificacion iBonificacion) {
    double d = iBonificacion.calcularBonificacion();
    System.out.println("Bonificación: " + d);
}
```

Y en `main`:

```java
EmpleadoPermanente ep = new EmpleadoPermanente();
EmpleadoGerente eg = new EmpleadoGerente();
Accionista acc = new Accionista();
EmpleadoTemporal et = new EmpleadoTemporal(); // no implementa interfaz

TratamientoBonificaciones(ep);
TratamientoBonificaciones(eg);
TratamientoBonificaciones(acc);
// TratamientoBonificaciones(et); ❌ Error: no implements IBonificacion
```

El profesor enfatiza que esto es **polimorfismo puro por interfaz**:

- No importa si el objeto es empleado o accionista.
- No importa si tiene salario o no.
- No importa su jerarquía.

Lo único que importa:

> “¿Implementa IBonificacion?”

Si la respuesta es sí → entra en el sistema.

Este diseño es escalable, elegante y mantenible.

5.6.8 Enseñanza profesional: interfaces como conectores universales

El profesor explica que en la práctica profesional:

- Interfaces = contratos
- Clases abstractas = plantillas
- Herencia = identidad
- Polimorfismo = flexibilidad
- Diseño limpio = separación de responsabilidades

Las interfaces:

- reducen acoplamiento,
- facilitan test unitarios,
- hacen que el código sea intercambiable,
- permiten reemplazar implementaciones sin modificar el resto,
- posibilitan programar “contra abstractions, not concretions” (principio de OOP moderno).

Y remata con una reflexión:

> “Las interfaces unen lo que no pertenece a la misma familia.
> Son el pegamento del diseño orientado a objetos moderno.”

5.7 Polimorfismo mediante interfaz — Tratamiento conjunto de objetos distintos gracias a un comportamiento común

Una vez creada la interfaz `IBonificacion`, el profesor demuestra cómo su uso permite lograr un tipo de polimorfismo mucho más flexible que el polimorfismo basado únicamente en herencia.

Este apartado distingue:

- **Polimorfismo vertical (por herencia)**
  Cuando varias clases comparten una superclase común (`Empleado`).
- **Polimorfismo horizontal o transversal (por interfaz)**
  Cuando varias clases comparten un comportamiento, aunque *no tengan relación jerárquica* (`IBonificacion` en empleados y accionistas).

El profesor subraya que **este segundo tipo es el más importante en diseño moderno** (Android, Spring, JavaEE…).

5.7.1 Polimorfismo clásico: herencia común

Primero repasa el tipo de polimorfismo tradicional:

```
Empleado e1 = new EmpleadoGerente();
Empleado e2 = new EmpleadoPermanente();
Empleado e3 = new EmpleadoTemporal();

System.out.println(e1.calcularSalario());
System.out.println(e2.calcularSalario());
System.out.println(e3.calcularSalario());
```

Conceptos:

- Todas las variables son `Empleado`.
- Cada objeto ejecuta su propia versión del método `calcularSalario()`.
- Este comportamiento se llama **polimorfismo dinámico** (late binding).
- Es posible porque todas las clases comparten una misma **identidad** (`Empleado` → clase base).

El profesor aclara:

> “Este polimorfismo funciona dentro de una misma familia de clases.”

Este es el polimorfismo vertical.

5.7.2 Polimorfismo aún más flexible: interfaz común

Ahora aparece el concepto crucial:

> "Una interfaz permite tratar como iguales a objetos completamente distintos que comparten una capacidad."

Ejemplo:

```
IBonificacion b1 = new EmpleadoPermanente();
IBonificacion b2 = new EmpleadoGerente();
IBonificacion b3 = new Accionista();
```

Observaciones esenciales:

- `EmpleadoPermanente` y `EmpleadoGerente` tienen identidad común (*sí son empleados*).
- `Accionista` **no es empleado**, no tiene salario base, no tiene id, no tiene contrato.
- Pero **todos pueden ser tratados polimórficamente** gracias a la interfaz.

Esto demuestra la **independencia total entre jerarquía e interfaz**.

5.7.3 Método `TratamientoBonificaciones()` — El punto donde se ve todo claro

El profesor implementa un método que solo trabaja con la interfaz:

```
private static void TratamientoBonificaciones(IBonificacion iBonificacion) {

    double d = iBonificacion.calcularBonificacion();
    System.out.println("Bonificación: " + d);
}
```

Este método:

- no sabe qué objeto está recibiendo,
- no necesita saberlo,
- no le importa la jerarquía del objeto,
- JAMÁS se expone al tipo concreto,
- solo le importa que el objeto **sepa calcular la bonificación**.

Esto es el polimorfismo por interfaz en estado puro.

5.7.4 Llamada al método en el `main()` — El ejemplo completo del profesor

```
EmpleadoPermanente ep = new EmpleadoPermanente();
EmpleadoGerente eg = new EmpleadoGerente();
EmpleadoTemporal et = new EmpleadoTemporal(); // No tiene bonificación
Accionista acc = new Accionista();
```

```
TratamientoBonificaciones(ep);
TratamientoBonificaciones(eg);
//TratamientoBonificaciones(et); // ❌ Error: no implementa IBonificacion
TratamientoBonificaciones(acc);
```

Qué se demuestra:

1. Polimorfismo transversal totalmente funcional

El método acepta:

- un empleado permanente,
- un gerente,
- un accionista,

pero rechaza un temporal.

Esto es exactamente lo que debe pasar.

2. Seguridad en tiempo de compilación

Si el objeto no implementa la interfaz → ERROR inmediato.
No hay sorpresas en tiempo de ejecución.

3. Desacoplamiento total del diseño

El método no nombra ni conoce ninguna clase concreta.

4. Extensibilidad infinita

Puedes añadir 20 clases nuevas con bonificación y TODAS funcionarán sin cambiar una línea del código actual.

5. Mantenibilidad profesional

Si mañana cambia la lógica de bonificación:

- solo se modifica `calcularBonificacion()` en cada clase,
- no se toca `TratamientoBonificaciones()`.

Este es el tipo de diseño que se enseña en patrones SOLID e ingeniería avanzada.

5.7.5 Comparación directa entre los dos tipos de polimorfismo

Tipo	Requisito	Qué conecta	Ámbito	Ejemplo
Herencia	Superclase común	Identidad	Vertical	`Empleado e = new EmpleadoGerente()`
Interfaz	Comportamiento común	Capacidades	Horizontal	`IBonificacion b = new Accionista()`

Este cuadro es clave en la mentalidad profesional:

> La herencia describe *qué eres*;
> la interfaz describe *qué puedes hacer*.

5.7.6 Ejemplo didáctico del profesor: “El Director de Orquesta”

El profesor resume esto con una analogía:

- Piensa en una orquesta.

- Todos los instrumentos son distintos.
- No comparten la misma “clase”, pero sí comparten un comportamiento importante: **pueden sonar**.
- Por eso, lo común no es su identidad (cuerdas, viento, madera…), sino la interfaz:

```
interface ISonable {
    void sonar();
}
```

De este modo:

- Violín, Flauta, Tuba y Triángulo implementan `ISonable`.
- El director de orquesta (equivalente al método `TratamientoBonificaciones`) solo necesita llamar a `sonar()` sin saber qué instrumento es.

Exactamente igual:

- `IBonificacion` = comportamiento común.
- `TratamientoBonificaciones()` = director de orquesta.

5.7.7 Relación directa con Android: Listeners y Callbacks

Aquí el profesor hace la conexión más importante con Android:

La programación de Android está **llena de interfaces**, no de herencias:

```
OnClickListener
OnLongClickListener
TextWatcher
OnScrollListener
Runnable
Comparator
```

Cuando haces:

```
button.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        // comportamiento concreto
    }
});
```

Estás usando EXACTAMENTE el mismo mecanismo que:

```
TratamientoBonificaciones(ep);
```

Android no sabe si la clase que recibe:

- es una Activity,
- es un Fragment,
- es un objeto anónimo,

- es una clase interna,
- o un adaptador custom.

Solo le importa:

> "¿Implementa la interfaz requerida?"

Este es el corazón de Android y de toda arquitectura moderna de eventos.

5.7.8 Enseñanza final del profesor: polimorfismo como herramienta de arquitectura

El profesor remata este apartado con un mensaje claro:

> "El polimorfismo no es magia: es diseño.
> Y si aprendes a diseñar con interfaces, tu código será escalable, limpio y profesional."

Elementos que resume:

- La herencia organiza una jerarquía estructural.
- Las interfaces organizan comportamientos transversales.
- Un diseño elegante mezcla ambas herramientas.
- Las interfaces permiten añadir capacidades sin modificar la jerarquía.
- El polimorfismo permite escribir código genérico que funciona con objetos concretos.

5.8 Relación directa con Android — Por qué toda esta teoría es imprescindible para programar en móviles

En esta parte de la clase, el profesor realiza un puente crucial entre los conceptos puros de POO (clases abstractas, herencia, polimorfismo, interfaces) y el mundo real del desarrollo Android.

El objetivo es claro:

> "Todo lo que hemos visto hoy **existe** y **se usa constantemente** en Android.
> Si entiendes Empleado + IBonificacion, entiendes el modelo de Android."

Vamos a desgranar esto punto por punto.

5.8.1 Android es una inmensa jerarquía de clases

En Android, casi TODO está construido sobre grandes árboles de herencia.

Ejemplos profundos del profesor:

1. Todas las pantallas son Activities que extienden de una clase base abstracta

```
public abstract class Activity {
    protected void onCreate(Bundle savedInstanceState) { ... }
    protected void onStart() { ... }
    protected void onResume() { ... }
    ...
}
```

Observa:

- `Activity` NO se puede usar sola → es **abstracta**.
- No puedes hacer:

```
Activity a = new Activity(); // ❌ no permitido
```

Esto es EXACTAMENTE lo mismo que:

```
Empleado e = new Empleado(); // ❌ es abstracta
```

2. Todas las vistas (botones, textos, sliders…) heredan de `View`

```
public class Button extends TextView { ... }
public class TextView extends View { ... }
public class ImageView extends View { ... }
```

Esto crea un árbol gigante:

```
View
 ├── TextView
 │     ├── Button
 │     ├── EditText
 │     └── CheckBox
 ├── ImageView
 └── SurfaceView
```

Y de nuevo, es igual que:

```
Empleado
 ├── EmpleadoGerente
 ├── EmpleadoPermanente
 └── EmpleadoTemporal
```

3. Todos los adaptadores de listas comparten una superclase abstracta

```
public abstract class RecyclerView.Adapter<VH extends RecyclerView.ViewHolder> {
    public abstract int getItemCount();
    public abstract void onBindViewHolder(...);
}
```

No se puede instanciar directamente, igual que `Empleado`.

La abstracción está por todas partes.

5.8.2 Android usa interfaces constantemente, mucho más que herencia

El profesor hace énfasis en que Android **prefiere interfaces** frente a herencia para eventos, acciones y comportamientos.

Los ejemplos son cientos:

1. Cuando escuchas un click en un botón:

```
button.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        ...
    }
});
```

Aquí está ocurriendo:

- EXACTAMENTE lo mismo que con `IBonificacion`.
- El sistema Android no sabe qué clase recibe.
- Solo sabe que implementa esta interfaz:

```
public interface OnClickListener {
    void onClick(View v);
}
```

2. Cuando escuchas cambios en un texto (EditText):

```
editText.addTextChangedListener(new TextWatcher() {
    @Override
    public void afterTextChanged(Editable s) { ... }
});
```

`TextWatcher` es una interfaz con múltiples métodos abstractos.

3. Cuando quieres ejecutar código en un hilo:

```
Runnable r = new Runnable() {
    @Override
    public void run() {
        // código en segundo plano
    }
};
```

`Runnable` es otra interfaz.

5.8.3 Conexión directa: IBonificacion ≈ OnClickListener

El profesor explica esto con claridad:

- En Java: `IBonificacion` agrupa objetos por un comportamiento común (calcular bonificación).
- En Android: `OnClickListener` agrupa objetos por un comportamiento común (responder a un click).

Ambas situaciones son idénticas conceptualmente:

Caso Java	Caso Android
`EmpleadoPermanente implements IBonificacion`	`MainActivity implements OnClickListener`
Método obligatorio: `calcularBonificacion()`	Método obligatorio: `onClick(View v)`
Polimorfismo horizontal	Polimorfismo horizontal

Es la misma mecánica, solo aplicada a móviles.

5.8.4 Actividades y Fragments: clases abstractas y sobreescritura obligatoria

En Android, es obligatorio sobreescribir ciertos métodos:

```
@Override
protected void onCreate(Bundle savedInstanceState) { ... }
```

Esto es equivalente a:

```
@Override
public double calcularSalario() { ... }
```

Ambos:

- provienen de un método abstracto o plantilla,
- deben ser implementados por la subclase,
- definen el comportamiento específico del objeto.

El profesor subraya:

> "Si has entendido por qué `EmpleadoGerente` tiene que implementar `calcularSalario`, ya sabes por qué `MainActivity` tiene que implementar `onCreate`."

5.8.5 Interfaces en Android como mecanismos de desacoplamiento

Las interfaces en Android sirven para:

1. Manejar eventos

Listeners: click, scroll, texto, foco, gestos, sensor…

2. Comunicar componentes entre sí

Por ejemplo, un Fragment se comunica con su Activity mediante una interfaz.

```
public interface OnProductoSeleccionadoListener {
    void onProductoSeleccionado(Producto p);
}
```

3. Adaptadores flexibles

`Comparator`, `Filterable`, `Parcelable`, `LifecycleObserver`, etc.

4. Delegación de lógica

Se delega una acción en una clase externa sin importar su tipo real.

Este es EXACTAMENTE el concepto detrás de `IBonificacion`.

5.8.6 Diseño limpio (Clean Code) aplicado en Android

El profesor muestra cómo la separación:

- clase abstracta (identidad)
- interfaz (comportamiento)

produce un diseño limpio.

Ejemplo:

Identidad en Android

- `Activity`
- `Fragment`
- `View`
- `RecyclerView.Adapter`

Comportamientos opcionales (interfaces)

- `OnClickListener`
- `TextWatcher`
- `Runnable`
- `OnScrollChangeListener`
- `OnFocusChangeListener`

Esto evita herencias artificiales que ensuciarían el diseño.

5.8.7 Cómo se conecta todo con el ejemplo de la clase

El profesor hace un paralelismo directo:

En la práctica Java (Empleados)	En Android
`Empleado` (abstracto)	`Activity` (abstracta)
`EmpleadoPermanente`	`MainActivity`
`IBonificacion`	`OnClickListener`
`calcularBonificacion()`	`onClick(View v)`
Polimorfismo transversal	Listeners, callbacks, eventos
Especialización por herencia	Ciclo de vida de Activities
No ensuciar la herencia	Arquitectura de Android

El mensaje clave:

> “Si entiendes herencia e interfaces en Java,
> ya entiendes el 70% de cómo funciona Android.”

5.8.8 Conclusión del profesor: la POO es la base de Android

Toda la programación de Activities, Fragments, Adapters, sensores y vistas se basa en:

- clases abstractas,
- herencia,
- interfaces,
- polimorfismo dinámico.

La clase de empleados NO es un ejemplo trivial.
Es el modelo exacto que utiliza Android para construir su arquitectura.

El profesor concluye:

> “Android no inventa nuevas reglas.
> Simplemente usa la POO de Java de forma intensiva.
> Cuando domines esto, desarrollar en Android será mucho más natural.”

5.9 Conclusión del profesor sobre polimorfismo + interfaces

El ejemplo de `Empleado` + `IBonificacion` que hemos construido en Eclipse resume de forma perfecta cómo funcionan las interfaces en Java y por qué son esenciales, especialmente en entornos como Android.

En este ejercicio se observa que:

- `EmpleadoPermanente`,
- `EmpleadoGerente`,
- y `Accionista`

son **clases completamente distintas**, con jerarquías y responsabilidades diferentes, pero pueden ser tratadas de manera uniforme gracias a que **todas implementan el mismo comportamiento**: la capacidad de calcular una bonificación.

Este es el concepto central:

> **Una interfaz permite agrupar objetos no por lo que *son*, sino por lo que *pueden hacer*.**

A diferencia de la herencia clásica, que une clases bajo una identidad común (`Empleado`), las interfaces permiten unir clases bajo un **comportamiento común**, aunque no tengan ninguna relación estructural entre ellas.

Aplicación directa a Android

El profesor recalca que este patrón es el corazón del desarrollo Android.
En Android, la inmensa mayoría de interacciones se realizan mediante **interfaces**, no mediante herencia.

Ejemplos directos:

- `View.OnClickListener` → “algo que puede responder a un click”
- `TextWatcher` → “algo que puede reaccionar cuando un texto cambia”
- `Runnable` → “algo que puede ejecutarse en un hilo”
- `Callback` → “algo que puede actuar como punto de retorno en un proceso”

En todos estos casos:

- Android no sabe si el objeto es una `Activity`, un `Fragment`, una clase anónima o un helper.
- Tampoco necesita saber nada de su jerarquía (como ocurre con `Empleado`, `Accionista`, etc.).
- Solo necesita **que implemente la interfaz correcta**.

Esto reproduce exactamente el patrón que hemos visto:

```
TratamientoBonificaciones(IBonificacion objeto);
```

En términos de Android:

```
button.setOnClickListener(new View.OnClickListener() {
    @Override
```

```
    public void onClick(View v) { ... }
});
```

El paralelismo es directo:

Ejemplo de la clase	Ejemplo en Android
`IBonificacion`	`OnClickListener`
`calcularBonificacion()`	`onClick(View v)`
`Accionista`, `EmpleadoPermanente`	`Activity`, clases anónimas, Fragments
Polimorfismo por interfaz	Eventos, callbacks, listeners

Mensaje final del profesor

El profesor cierra esta parte explicando que las interfaces son indispensables porque:

- permiten escribir código genérico desacoplado,
- permiten añadir nuevas clases sin modificar las existentes,
- permiten trabajar con objetos distintos que comparten un comportamiento común,
- permiten mantener la jerarquía de clases limpia, sin métodos que no son universales,
- y permiten aplicar polimorfismo incluso entre clases que no comparten padre.

Su conclusión es directa:

> **“Las interfaces son la herramienta que hace posible conectar objetos distintos mediante un mismo comportamiento.**
> **En Android esto es fundamental: cada evento, callback o listener funciona exactamente igual que el ejemplo de IBonificacion.”**
